(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
21 May 2004 (21.05.2004)

PCT

(10) International Publication Number
WO 2004/042574 A2

(51) International Patent Classification[7]: G06F 9/46

(21) International Application Number: PCT/US2003/034559

(22) International Filing Date: 31 October 2003 (31.10.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/422,503	31 October 2002 (31.10.2002)	US
10/684,102	9 October 2003 (09.10.2003)	US
10/683,929	9 October 2003 (09.10.2003)	US
10/683,932	9 October 2003 (09.10.2003)	US
10/684,053	9 October 2003 (09.10.2003)	US
10/684,057	9 October 2003 (09.10.2003)	US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 60/422,503 (CIP)
Filed on 31 October 2002 (31.10.2002)

(71) Applicant *(for all designated States except US)*: LOCKHEED MARTIN CORPORATION [US/US]; Bldg 400, Mail Drop 043, 9500 Godwin Drive, Manassas, VA 20110 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: MATHUR, Chandan [US/US]; 11162 Privatess Court, Manassas, VA 20109 (US). HELLENBACH, Scott [US/US]; 15381 Quail Ridge Drive, Amessville, VA 20106 (US). RAPP, John, W. [US/US]; 9350 River Crest Road, Manassas, VA 20110 (US).

(74) Agents: SANTARELLI, Bryan, A. et al.; Graybeal Jackson Haley LLP, Suite 350, 155-108th Avenue NE, Bellevue, WA 98004 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (BW, GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 2004/042574 A2

(54) Title: COMPUTING MACHINE HAVING IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD

(57) Abstract: A computing machine includes a first buffer and a processor coupled to the buffer. The processor executes an application, a first data-transfer object, and a second data-transfer object, publishes data under the control of the application, loads the published data into the buffer under the control of the first data-transfer object, and retrieves the published data from the buffer under the control of the second data-transfer object. Alternatively, the processor retrieves data and loads the retrieved data into the buffer under the control of the first data-transfer object, unloads the data from the buffer under the control of the second data-transfer object, and processes the unloaded data under the control of the application. Where the computing machine is a peer-vector machine that includes a hardwired pipeline accelerator coupled to the processor, the buffer and data-transfer objects facilitate the transfer of data between the application and the accelerator.

COMPUTING MACHINE HAVING IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD

CLAIM OF PRIORITY

[1] This application claims priority to U.S. Provisional Application Serial No. 60/422,503, filed on October 31, 2002, which is incorporated by reference.

CROSS REFERENCE TO RELATED APPLICATIONS

[2] This application is related to U.S. Patent App. Serial Nos. 10/684,102 entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD; 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD; 10/684,057 entitled PROGRAMMABLE CIRCUIT AND RELATED COMPUTING MACHINE AND METHOD; and 10/683,932 entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD; all filed on October 9, 2003, and having a common owner, and which are incorporated by reference.

BACKGROUND

[3] A common computing architecture for processing relatively large amounts of data in a relatively short period of time includes multiple interconnected processors that share the processing burden. By sharing the processing burden, these multiple processors can often process the data more quickly than a single processor can for a given clock frequency. For example, each of the processors can process a respective portion of the data or execute a respective portion of a processing algorithm.

[4] **FIG. 1** is a schematic block diagram of a conventional computing machine ***10*** having a multi-processor architecture. The machine ***10*** includes a master processor ***12*** and coprocessors $14_1 - 14_n$, which communicate with each other and the master processor via a bus ***16***, an input port ***18*** for receiving raw data from a remote device (not shown in **FIG. 1**), and an output port ***20*** for providing processed data to the remote source. The machine ***10*** also includes a memory ***22*** for the master processor ***12***, respective memories $24_1 - 24_n$ for the coprocessors 14_1

– ***14$_n$***, and a memory ***26*** that the master processor and coprocessors share via the bus ***16***. The memory ***22*** serves as both a program and a working memory for the master processor ***12***, and each memory ***24$_1$*** – ***24$_n$*** serves as both a program and a working memory for a respective coprocessor ***14$_1$*** – ***14$_n$***. The shared memory ***26*** allows the master processor ***12*** and the coprocessors ***14*** to transfer data among themselves, and from/to the remote device via the ports ***18*** and ***20***, respectively. The master processor ***12*** and the coprocessors ***14*** also receive a common clock signal that controls the speed at which the machine ***10*** processes the raw data.

[5] In general, the computing machine ***10*** effectively divides the processing of raw data among the master processor ***12*** and the coprocessors ***14***. The remote source (not shown in **FIG.** 1) such as a sonar array loads the raw data via the port ***18*** into a section of the shared memory ***26***, which acts as a first-in-first-out (FIFO) buffer (not shown) for the raw data. The master processor ***12*** retrieves the raw data from the memory ***26*** via the bus ***16***, and then the master processor and the coprocessors ***14*** process the raw data, transferring data among themselves as necessary via the bus ***16***. The master processor ***12*** loads the processed data into another FIFO buffer (not shown) defined in the shared memory ***26***, and the remote source retrieves the processed data from this FIFO via the port ***20***.

[6] In an example of operation, the computing machine ***10*** processes the raw data by sequentially performing n + 1 respective operations on the raw data, where these operations together compose a processing algorithm such as a Fast Fourier Transform (FFT). More specifically, the machine ***10*** forms a data-processing pipeline from the master processor ***12*** and the coprocessors ***14***. For a given frequency of the clock signal, such a pipeline often allows the machine ***10*** to process the raw data faster than a machine having only a single processor.

[7] After retrieving the raw data from the raw-data FIFO (not shown) in the memory ***26***, the master processor ***12*** performs a first operation, such as a trigonometric function, on the raw data. This operation yields a first result, which the processor ***12*** stores in a first-result FIFO (not shown) defined within the memory ***26***. Typically, the processor ***12*** executes a program stored in the memory ***22***, and performs the above-described actions under the control of the program. The

processor **12** may also use the memory **22** as working memory to temporarily store data that the processor generates at intermediate intervals of the first operation.

[8] Next, after retrieving the first result from the first-result FIFO (not shown) in the memory **26**, the coprocessor **14_1** performs a second operation, such as a logarithmic function, on the first result. This second operation yields a second result, which the coprocessor **14_1** stores in a second-result FIFO (not shown) defined within the memory **26**. Typically, the coprocessor **14_1** executes a program stored in the memory **24_1**, and performs the above-described actions under the control of the program. The coprocessor **14_1** may also use the memory **24_1** as working memory to temporarily store data that the coprocessor generates at intermediate intervals of the second operation.

[9] Then, the coprocessors **$24_2 - 24_n$** sequentially perform third – n^{th} operations on the second – $(n-1)^{th}$ results in a manner similar to that discussed above for the coprocessor **24_1**.

[10] The n^{th} operation, which is performed by the coprocessor **24_n**, yields the final result, *i.e.*, the processed data. The coprocessor **24_n** loads the processed data into a processed-data FIFO (not shown) defined within the memory **26**, and the remote device (not shown in **FIG. 1**) retrieves the processed data from this FIFO.

[11] Because the master processor **12** and coprocessors **14** are simultaneously performing different operations of the processing algorithm, the computing machine **10** is often able to process the raw data faster than a computing machine having a single processor that sequentially performs the different operations. Specifically, the single processor cannot retrieve a new set of the raw data until it performs all n + 1 operations on the previous set of raw data. But using the pipeline technique discussed above, the master processor **12** can retrieve a new set of raw data after performing only the first operation. Consequently, for a given clock frequency, this pipeline technique can increase the speed at which the machine **10** processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

[12] Alternatively, the computing machine **10** may process the raw data in parallel by simultaneously performing n + 1 instances of a processing algorithm,

such as an FFT, on the raw data. That is, if the algorithm includes n + 1 sequential operations as described above in the previous example, then each of the master processor ***12*** and the coprocessors ***14*** sequentially perform all n + 1 operations on respective sets of the raw data. Consequently, for a given clock frequency, this parallel-processing technique, like the above-described pipeline technique, can increase the speed at which the machine ***10*** processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

[13] Unfortunately, although the computing machine ***10*** can process data more quickly than a single-processor computer machine (not shown in **FIG. 1**), the data-processing speed of the machine ***10*** is often significantly less than the frequency of the processor clock. Specifically, the data-processing speed of the computing machine ***10*** is limited by the time that the master processor ***12*** and coprocessors ***14*** require to process data. For brevity, an example of this speed limitation is discussed in conjunction with the master processor ***12***, although it is understood that this discussion also applies to the coprocessors ***14***. As discussed above, the master processor ***12*** executes a program that controls the processor to manipulate data in a desired manner. This program includes a sequence of instructions that the processor ***12*** executes. Unfortunately, the processor ***12*** typically requires multiple clock cycles to execute a single instruction, and often must execute multiple instructions to process a single value of data. For example, suppose that the processor ***12*** is to multiply a first data value A (not shown) by a second data value B (not shown). During a first clock cycle, the processor ***12*** retrieves a multiply instruction from the memory ***22***. During second and third clock cycles, the processor ***12*** respectively retrieves A and B from the memory ***26***. During a fourth clock cycle, the processor ***12*** multiplies A and B, and, during a fifth clock cycle, stores the resulting product in the memory ***22*** or ***26*** or provides the resulting product to the remote device (not shown). This is a best-case scenario, because in many cases the processor ***12*** requires additional clock cycles for overhead tasks such as initializing and closing counters. Therefore, at best the processor ***12*** requires five clock cycles, or an average of 2.5 clock cycles per data value, to process A and B..

[14] Consequently, the speed at which the computing machine ***10*** processes data is often significantly lower than the frequency of the clock that drives the master processor ***12*** and the coprocessors ***14***. For example, if the processor ***12*** is clocked at 1.0 Gigahertz (GHz) but requires an average of 2.5 clock cycles per data value, then the effective data-processing speed equals (1.0 GHz)/2.5 = 0.4 GHz. This effective data-processing speed is often characterized in units of operations per second. Therefore, in this example, for a clock speed of 1.0 GHz, the processor ***12*** would be rated with a data-processing speed of 0.4 Gigaoperations/second (Gops).

[15] **FIG. 2** is a block diagram of a hardwired data pipeline ***30*** that can typically process data faster than a processor can for a given clock frequency, and often at substantially the same rate at which the pipeline is clocked. The pipeline ***30*** includes operator circuits $\boldsymbol{32_1 - 32_n}$ that each perform a respective operation on respective data without executing program instructions. That is, the desired operation is "burned in" to a circuit ***32*** such that it implements the operation automatically, without the need of program instructions. By eliminating the overhead associated with executing program instructions, the pipeline ***30*** can typically perform more operations per second than a processor can for a given clock frequency.

[16] For example, the pipeline ***30*** can often solve the following equation faster than a processor can for a given clock frequency:

$$Y(x_k) = (5x_k + 3)2^{xk}$$

where x_k represents a sequence of raw data values. In this example, the operator circuit $\boldsymbol{32_1}$ is a multiplier that calculates $5x_k$, the circuit $\boldsymbol{32_2}$ is an adder that calculates $5x_k + 3$, and the circuit $\boldsymbol{32_n}$ (n = 3) is a multiplier that calculates $(5x_k + 3)2^{xk}$.

[17] During a first clock cycle k=1, the circuit $\boldsymbol{32_1}$ receives data value x_1 and multiplies it by 5 to generate $5x_1$.

[18] During a second clock cycle k = 2, the circuit $\boldsymbol{32_2}$ receives $5x_1$ from the circuit $\boldsymbol{32_1}$ and adds 3 to generate $5x_1 + 3$. Also, during the second clock cycle, the circuit $\boldsymbol{32_1}$ generates $5x_2$.

[19] [19] During a third clock cycle k = 3, the circuit $\boldsymbol{32_3}$ receives $5x_1 + 3$ from the circuit $\boldsymbol{32_2}$ and multiplies by 2^{x1}(effectively left shifts $5x_1 + 3$ by x_1) to

generate the first result $(5x_1 + 3)2^{x1}$. Also during the third clock cycle, the circuit ***32₁*** generates $5x_3$ and the circuit ***32₂*** generates $5x_2 + 3$.

[20] The pipeline ***30*** continues processing subsequent raw data values x_k in this manner until all the raw data values are processed.

[21] Consequently, a delay of two clock cycles after receiving a raw data value x_1 — this delay is often called the latency of the pipeline ***30*** — the pipeline generates the result $(5x_1 + 3)2^{x1}$, and thereafter generates one result — *e.g.*, $(5x_2 + 3)2^{x2}$, $(5x_3 + 3)2^{x3}$, . . ., $5x_n + 3)2^{xn}$ — each clock cycle.

[22] Disregarding the latency, the pipeline ***30*** thus has a data-processing speed equal to the clock speed. In comparison, assuming that the master processor ***12*** and coprocessors ***14*** (**FIG. 1**) have data-processing speeds that are 0.4 times the clock speed as in the above example, the pipeline ***30*** can process data 2.5 times faster than the computing machine ***10*** (**FIG. 1**) for a given clock speed.

[23] Still referring to **FIG. 2**, a designer may choose to implement the pipeline ***30*** in a programmable logic IC (PLIC), such as a field-programmable gate array (FPGA), because a PLIC allows more design and modification flexibility than does an application specific IC (ASIC). To configure the hardwired connections within a PLIC, the designer merely sets interconnection-configuration registers disposed within the PLIC to predetermined binary states. The combination of all these binary states is often called "firmware." Typically, the designer loads this firmware into a nonvolatile memory (not shown in **FIG. 2**) that is coupled to the PLIC. When one "turns on" the PLIC, it downloads the firmware from the memory into the interconnection-configuration registers. Therefore, to modify the functioning of the PLIC, the designer merely modifies the firmware and allows the PLIC to download the modified firmware into the interconnection-configuration registers. This ability to modify the PLIC by merely modifying the firmware is particularly useful during the prototyping stage and for upgrading the pipeline ***30*** "in the field".

[24] Unfortunately, the hardwired pipeline ***30*** typically cannot execute all algorithms, particularly those that entail significant decision making. A processor can typically execute a decision-making instruction (*e.g.*, conditional instructions such as "if A, then go to B, else go to C") approximately as fast as it can execute an

operational instruction (*e.g.*, "A + B") of comparable length. But although the pipeline ***30*** may be able to make a relatively simple decision (*e.g.*, "A > B?"), it typically cannot execute a relatively complex decision (*e.g.*, "if A, then go to B, else go to C"). And although one may be able to design the pipeline ***30*** to execute such a complex decision, the size and complexity of the required circuitry often makes such a design impractical, particularly where an algorithm includes multiple different complex decisions.

[25] Consequently, processors are typically used in applications that require significant decision making, and hardwired pipelines are typically limited to "number crunching" applications that entail little or no decision making.

[26] Furthermore, as discussed below, it is typically much easier for one to design/modify a processor-based computing machine, such as the computing machine ***10*** of **FIG. 1**, than it is to design/modify a hardwired pipeline such as the pipeline ***30*** of **FIG. 2**, particularly where the pipeline ***30*** includes multiple PLICs.

[27] Computing components, such as processors and their peripherals (*e.g.*, memory), typically include industry-standard communication interfaces that facilitate the interconnection of the components to form a processor-based computing machine.

[28] Typically, a standard communication interface includes two layers: a physical layer and a service layer.

[29] The physical layer includes the circuitry and the corresponding circuit interconnections that form the interface and the operating parameters of this circuitry. For example, the physical layer includes the pins that connect the component to a bus, the buffers that latch data received from the pins, and the drivers that drive data onto the pins. The operating parameters include the acceptable voltage range of the data signals that the pins receive, the signal timing for writing and reading data, and the supported modes of operation (*e.g.*, burst mode, page mode). Conventional physical layers include transistor-transistor logic (TTL) and RAMBUS.

[30] The service layer includes the protocol by which a computing component transfers data. The protocol defines the format of the data and the

manner in which the component sends and receives the formatted data. Conventional communication protocols include file-transfer protocol (FTP) and TCP/IP **(expand).**

[31] Consequently, because manufacturers and others typically design computing components having industry-standard communication interfaces, one can typically design the interface of such a component and interconnect it to other computing components with relatively little effort. This allows one to devote most of his time to designing the other portions of the computing machine, and to easily modify the machine by adding or removing components.

[32] Designing a computing component that supports an industry-standard communication interface allows one to save design time by using an existing physical-layer design from a design library. This also insures that he/she can easily interface the component to off-the-shelf computing components.

[33] And designing a computing machine using computing components that support a common industry-standard communication interface allows the designer to interconnect the components with little time and effort. Because the components support a common interface, the designer can interconnect them via a system bus with little design effort. And because the supported interface is an industry standard, one can easily modify the machine. For example, one can add different components and peripherals to the machine as the system design evolves, or can easily add/design next-generation components as the technology evolves. Furthermore, because the components support a common industry-standard service layer, one can incorporate into the computing machine's software an existing software module that implements the corresponding protocol. Therefore, one can interface the components with little effort because the interface design is essentially already in place, and thus can focus on designing the portions (*e.g.*, software) of the machine that cause the machine to perform the desired function(s).

[34] But unfortunately, there are no known industry-standard communication interfaces for components, such as PLICs, used to form hardwired pipelines such as the pipeline ***30*** of **FIG. 2**.

[35] Consequently, to design a pipeline having multiple PLICs, one typically spends a significant amount of time and exerts a significant effort designing and debugging the communication interface between the PLICs "from scratch." Typically, such an ad hoc communication interface depends on the parameters of the data being transferred between the PLICs. Likewise, to design a pipeline that interfaces to a processor, one would have to spend a significant amount of time and exert a significant effort in designing and debugging the communication interface between the pipeline and the processor from scratch.

[36] Similarly, to modify such a pipeline by adding a PLIC to it, one typically spends a significant amount of time and exerts a significant effort designing and debugging the communication interface between the added PLIC and the existing PLICs. Likewise, to modify a pipeline by adding a processor, or to modify a computing machine by adding a pipeline, one would have to spend a significant amount of time and exert a significant effort in designing and debugging the communication interface between the pipeline and processor.

[37] Consequently, referring to **FIGS. 1** and **2**, because of the difficulties in interfacing multiple PLICs and in interfacing a processor to a pipeline, one is often forced to make significant tradeoffs when designing a computing machine. For example, with a processor-based computing machine, one is forced to trade number-crunching speed and design/modification flexibility for complex decision-making ability. Conversely, with a hardwired pipeline-based computing machine, one is forced to trade complex-decision-making ability and design/modification flexibility for number-crunching speed. Furthermore, because of the difficulties in interfacing multiple PLICs, it is often impractical for one to design a pipeline-based machine having more than a few PLICs. As a result, a practical pipeline-based machine often has limited functionality. And because of the difficulties in interfacing a processor to a PLIC, it would be impractical to interface a processor to more than one PLIC. As a result, the benefits obtained by combining a processor and a pipeline would be minimal.

[38] Therefore, a need has arisen for a new computing architecture that allows one to combine the decision-making ability of a processor-based machine with the number-crunching speed of a hardwired-pipeline-based machine.

SUMMARY

[39] In an embodiment of the invention, a computing machine includes a first buffer and a processor coupled to the buffer. The processor is operable to execute an application, a first data-transfer object, and a second data-transfer object, publish data under the control of the application, load the published data into the buffer under the control of the first data-transfer object, and retrieve the published data from the buffer under the control of the second data-transfer object.

[40] According to another embodiment of the invention, the processor is operable to retrieve data and load the retrieved data into the buffer under the control of the first data-transfer object, unload the data from the buffer under the control of the second data-transfer object, and process the unloaded data under the control of the application.

[41] Where the computing machine is a peer-vector machine that includes a hardwired pipeline accelerator coupled to the processor, the buffer and data-transfer objects facilitate the transfer of data — whether unidirectional or bidirectional — between the application and the accelerator.

BRIEF DESCRIPTION OF THE DRAWINGS

[42] **FIG. 1** is a block diagram of a computing machine having a conventional multi-processor architecture.

[43] **FIG. 2** is a block diagram of a conventional hardwired pipeline.

[44] **FIG. 3** is schematic block diagram of a computing machine having a peer-vector architecture according to an embodiment of the invention.

[45] **FIG. 4** is a functional block diagram of the host processor of **FIG. 3** according to an embodiment of the invention.

[46] **FIG. 5** is a functional block diagram of the data-transfer paths between the data-processing application and the pipeline bus of **FIG. 4** according to an embodiment of the invention.

[47] **FIG. 6** is a functional block diagram of the data-transfer paths between the accelerator exception manager and the pipeline bus of **FIG. 4** according to an embodiment of the invention.

[48] **FIG. 7** is a functional block diagram of the data-transfer paths between the accelerator configuration manager and the pipeline bus of **FIG. 4** according to an embodiment of the invention.

DETAILED DESCRIPTION

[49] **FIG. 3** is a schematic block diagram of a computing machine ***40***, which has a peer-vector architecture according to an embodiment of the invention. In addition to a host processor ***42***, the peer-vector machine ***40*** includes a pipeline accelerator ***44***, which performs at least a portion of the data processing, and which thus effectively replaces the bank of coprocessors ***14*** in the computing machine ***10*** of **FIG. 1**. Therefore, the host-processor ***42*** and the accelerator ***44*** are "peers" that can transfer data vectors back and forth. Because the accelerator ***44*** does not execute program instructions, it typically performs mathematically intensive operations on data significantly faster than a bank of coprocessors can for a given clock frequency. Consequently, by combing the decision-making ability of the processor ***42*** and the number-crunching ability of the accelerator ***44***, the machine ***40*** has the same abilities as, but can often process data faster than, a conventional computing machine such as the machine ***10***. Furthermore, as discussed below and in previously cited U.S. Patent App. Serial No. 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, providing the accelerator ***44*** with the same communication interface as the host processor ***42*** facilitates the design and modification of the machine ***40***, particularly where the communications interface is an industry standard. And where the accelerator ***44*** includes multiple components (*e.g.*, PLICs), providing these components with this same communication interface facilitates the design and modification of the accelerator, particularly where the communication interface is an industry standard. Moreover, the machine ***40*** may also provide other advantages as described below and in the previously cited patent applications.

[50] Still referring to **FIG. 3**, in addition to the host processor ***42*** and the pipeline accelerator ***44***, the peer-vector computing machine ***40*** includes a processor memory ***46***, an interface memory ***48***, a bus ***50***, a firmware memory ***52***, optional raw-data input ports ***54*** and ***56***, processed-data output ports ***58*** and ***60***, and an optional router ***61***.

[51] The host processor ***42*** includes a processing unit ***62*** and a message handler ***64***, and the processor memory ***46*** includes a processing-unit memory ***66*** and a handler memory ***68***, which respectively serve as both program and working memories for the processor unit and the message handler. The processor memory ***46*** also includes an accelerator-configuration registry ***70*** and a message-configuration registry ***72***, which store respective configuration data that allow the host processor ***42*** to configure the functioning of the accelerator ***44*** and the structure of the messages that the message handler ***64*** sends and receives.

[52] The pipeline accelerator ***44*** is disposed on at least one PLIC (not shown) and includes hardwired pipelines $\boldsymbol{74_1} - \boldsymbol{74_n}$, which process respective data without executing program instructions. The firmware memory ***52*** stores the configuration firmware for the accelerator ***44***. If the accelerator ***44*** is disposed on multiple PLICs, these PLICs and their respective firmware memories may be disposed on multiple circuit boards, *i.e.*, daughter cards (not shown). The accelerator ***44*** and daughter cards are discussed further in previously cited U.S. Patent App. Serial Nos. 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD and 10/683,932 entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD. Alternatively, the accelerator ***44*** may be disposed on at least one ASIC, and thus may have internal interconnections that are unconfigurable. In this alternative, the machine ***40*** may omit the firmware memory ***52***. Furthermore, although the accelerator ***44*** is shown including multiple pipelines ***74***, it may include only a single pipeline. In addition, although not shown, the accelerator ***44*** may include one or more processors such as a digital-signal processor (DSP).

[53] The general operation of the peer-vector machine ***40*** is discussed in previously cited U.S. Patent App. Serial No. 10/684,102 entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, and the functional topology and operation of the host processor ***42*** is discussed below in conjunction with **FIGS. 4 – 7**. **FIG. 4** is a functional block diagram of the host processor ***42*** and the pipeline bus ***50*** of **FIG. 3** according to an embodiment of the invention. Generally, the processing unit ***62*** executes one or more software

applications, and the message handler ***64*** executes one or more software objects that transfer data between the software application(s) and the pipeline accelerator ***44*** (**FIG. 3**). Splitting the data-processing, data-transferring, and other functions among different applications and objects allows for easier design and modification of the host-processor software. Furthermore, although in the following description a software application is described as performing a particular operation, it is understood that in actual operation, the processing unit ***62*** or message handler ***64*** executes the software application and performs this operation under the control of the application. Likewise, although in the following description a software object is described as performing a particular operation, it is understood that in actual operation, the processing unit ***62*** or message handler ***64*** executes the software object and performs this operation under the control of the object.

[54] Still referring to **FIG. 4**, the processing unit ***62*** executes a data-processing application ***80***, an accelerator exception manager application (hereinafter the exception manager) ***82***, and an accelerator configuration manager application (hereinafter the configuration manager) ***84***, which are collectively referred to as the processing-unit applications. The data-processing application processes data in cooperation with the pipeline accelerator ***44*** (**FIG. 3**). For example, the data-processing application ***80*** may receive raw sonar data via the port ***54*** (**FIG. 3**), parse the data, and send the parsed data to the accelerator ***44***, and the accelerator may perform an FFT on the parsed data and return the processed data to the data-processing application for further processing. The exception manager ***82*** handles exception messages from the accelerator ***44***, and the configuration manager ***84*** loads the accelerator's configuration firmware into the memory ***52*** during initialization of the peer-vector machine ***40*** (**FIG. 3**). The configuration manager ***84*** may also reconfigure the accelerator ***44*** after initialization in response to, *e.g.*, a malfunction of the accelerator. As discussed further below in conjunction with **FIGS. 6 – 7**, the processing-unit applications may communicate with each other directly as indicated by the dashed lines ***85***, ***87***, and ***89***, or may communicate with each other via the data-transfer objects ***86***. The message handler ***64*** executes the data-transfer objects ***86***, a communication object ***88***, and input and output read objects ***90*** and ***92***, and may execute input and output queue objects ***94*** and ***96***. The data-transfer

objects ***86*** transfer data between the communication object ***88*** and the processing-unit applications, and may use the interface memory ***48*** as a data buffer to allow the processing-unit applications and the accelerator ***44*** to operate independently. For example, the memory ***48*** allows the accelerator ***44***, which is often faster than the data-processing application ***80***, to operate without "waiting" for the data-processing application. The communication object ***88*** transfers data between the data objects ***86*** and the pipeline bus ***50***. The input and output read objects ***90*** and ***92*** control the data-transfer objects ***86*** as they transfer data between the communication object ***88*** and the processing-unit applications. And, when executed, the input and output queue objects ***94*** and ***96*** cause the input and output read objects ***90*** and ***92*** to synchronize this transfer of data according to a desired priority

[55] Furthermore, during initialization of the peer-vector machine ***40*** (**FIG. 3**), the message handler ***64*** instantiates and executes a conventional object factory ***98***, which instantiates the data-transfer objects ***86*** from configuration data stored in the message-configuration registry ***72*** (**FIG. 3**). The message handler ***64*** also instantiates the communication object ***88***, the input and output reader objects ***90*** and ***92***, and the input and output queue objects ***94*** and ***96*** from the configuration data stored in the message-configuration registry ***72***. Consequently, one can design and modify these software objects, and thus their data-transfer parameters, by merely designing or modifying the configuration data stored in the registry ***72***. This is typically less time consuming than designing or modifying each software object individually.

[56] The operation of the host processor ***42*** of **FIG. 4** is discussed below in conjunction with **FIGS. 5 – 7**.

<u>Data Processing</u>

[57] **FIG. 5** is a functional block diagram of the data-processing application ***80***, the data-transfer objects ***86***, and the interface memory ***48*** of **FIG. 4** according to an embodiment of the invention.

[58] The data-processing application ***80*** includes a number of threads ***100_1*** – ***100_n***, which each perform a respective data-processing operation. For example,

the thread 100_1 may perform an addition, and the thread 100_2 may perform a subtraction, or both the threads 100_1 and 100_2 may perform an addition.

[59] Each thread **100** generates, *i.e.*, publishes, data destined for the pipeline accelerator **44** (**FIG. 3**), receives, *i.e.*, subscribes to, data from the accelerator, or both publishes and subscribes to data. For example, each of the threads 100_1 - 100_4 both publish and subscribe to data from the accelerator **44**. A thread **100** may also communicate directly with another thread **100**. For example, as indicated by the dashed line **102**, the threads 100_3 and 100_4 may directly communicate with each other. Furthermore, a thread **100** may receive data from or send data to a component (not shown) other than the accelerator **44** (**FIG. 3**). But for brevity, discussion of data transfer between the threads **100** and such another component is omitted.

[60] Still referring to **FIG. 5**, the interface memory **48** and the data-transfer objects 86_{1a} – 86_{nb} functionally form a number of unidirectional channels 104_1 – 104_n for transferring data between the respective threads **100** and the communication object **88**. The interface memory **48** includes a number of buffers 106_1 – 106_n, one buffer per channel **104**. The buffers **106** may each hold a single grouping (*e.g.*, byte, word, block) of data, or at least some of the buffers may be FIFO buffers that can each store respective multiple groupings of data. There are also two data objects **86** per channel **104**, one for transferring data between a respective thread **100** and a respective buffer **106**, and the other for transferring data between the buffer **106** and the communication object **88**. For example, the channel 104_1 includes a buffer 106_1, a data-transfer object 86_{1a} for transferring published data from the thread 100_1 to the buffer 106_1, and a data-transfer object 86_{1b} for transferring the published data from the buffer 106_1 to the communication object **88**. Including a respective channel **104** for each allowable data transfer reduces the potential for data bottlenecks and also facilitates the design and modification of the host processor **42** (**FIG. 4**).

[61] Referring to **FIGS. 3 - 5**, the operation of the host processor **42** during its initialization and while executing the data-processing application **80**, the data-transfer objects **86**, the communication object **88**, and the optional reader and queue objects **90, 92**, **94**, and **96** is discussed according to an embodiment of the invention.

[62] During initialization of the host processor **42**, the object factory **98** instantiates the data-transfer objects **86** and defines the buffers **104**. Specifically, the object factory **98** downloads the configuration data from the registry **72** and generates the software code for each data-transfer object $\boldsymbol{86_{xb}}$ that the data-processing application **80** may need. The identity of the data-transfer objects $\boldsymbol{86_{xb}}$ that the application **80** may need is typically part of the configuration data — the application **80**, however, need not use all of the data-transfer objects **86**. Then, from the generated objects $\boldsymbol{86_{xb}}$, the object factory **98** respectively instantiates the data objects $\boldsymbol{86_{xa}}$. Typically, as discussed in the example below, the object factory **98** instantiates data-transfer objects $\boldsymbol{86_{xa}}$ and $\boldsymbol{86_{xb}}$ that access the same buffer **104** as multiple instances of the same software code. This reduces the amount of code that the object factory **98** would otherwise generate by approximately one half. Furthermore, the message handler **64** may determine which, if any, data-transfer objects **86** the application **80** does not need, and delete the instances of these unneeded data-transfer objects to save memory. Alternatively, the message handler **64** may make this determination before the object factory **98** generates the data-transfer objects **86**, and cause the object factory to instantiate only the data-transfer objects that the application **80** needs. In addition, because the data-transfer objects **86** include the addresses of the interface memory **48** where the respective buffers **104** are located, the object factory **98** effectively defines the sizes and locations of the buffers when it instantiates the data-transfer objects.

[63] For example, the object factory **98** instantiates the data-transfer objects $\boldsymbol{86_{1a}}$ and $\boldsymbol{86_{1b}}$ in the following manner. First, the factory **98** downloads the configuration data from the registry **72** and generates the common software code for the data-transfer object $\boldsymbol{86_{1a}}$ and $\boldsymbol{86_{1b}}$. Next, the factory **98** instantiates the data-transfer objects $\boldsymbol{86_{1a}}$ and $\boldsymbol{86_{1b}}$ as respective instances of the common software code. That is, the message handler **64** effectively copies the common software code to two locations of the handler memory **68** or to other program memory (not shown), and executes one location as the object $\boldsymbol{86_{1a}}$ and the other location as the object $\boldsymbol{86_{1b}}$.

[64] Still referring to **FIGS. 3-5**, after initialization of the host processor ***42***, the data-processing application ***80*** processes data and sends data to and receives data from the pipeline accelerator ***44***.

[65] An example of the data-processing application ***80*** sending data to the accelerator ***44*** is discussed in conjunction with the channel $\mathbf{104_1}$.

[66] First, the thread $\mathbf{100_1}$ generates and publishes data to the data-transfer object $\mathbf{86_{1a}}$. The thread $\mathbf{100_1}$ may generate the data by operating on raw data that it receives from the accelerator ***44*** (further discussed below) or from another source (not shown) such as a sonar array or a data base via the port ***54***.

[67] Then, the data-object $\mathbf{86_{1a}}$ loads the published data into the buffer $\mathbf{106_1}$.

[68] Next, the data-transfer object $\mathbf{86_{1b}}$ determines that the buffer $\mathbf{106_1}$ has been loaded with newly published data from the data-transfer object $\mathbf{86_{1a}}$. The output reader object ***92*** may periodically instruct the data-transfer object $\mathbf{86_{1b}}$ to check the buffer $\mathbf{106_1}$ for newly published data. Alternatively, the output reader object ***92*** notifies the data-transfer object $\mathbf{86_{1b}}$ when the buffer $\mathbf{106_1}$ has received newly published data. Specifically, the output queue object ***96*** generates and stores a unique identifier (not shown) in response to the data-transfer object $\mathbf{86_{1a}}$ storing the published data in the buffer $\mathbf{106_1}$. In response to this identifier, the output reader object ***92*** notifies the data-transfer object $\mathbf{86_{1b}}$ that the buffer $\mathbf{106_1}$ contains newly published data. Where multiple buffers ***106*** contain respective newly published data, then the output queue object ***96*** may record the order in which this data was published, and the output reader object ***92*** may notify the respective data-transfer objects $\mathbf{86_{xb}}$ in the same order. Thus, the output reader object ***92*** and the output queue object ***96*** synchronize the data transfer by causing the first data published to be the first data that the respective data-transfer object $\mathbf{86_{xb}}$ sends to the accelerator ***44***, the second data published to be the second data that the respective data-transfer object $\mathbf{86_{xb}}$ sends to the accelerator, etc. In another alternative where multiple buffers ***106*** contain respective newly published data, the output reader and output queue objects ***92*** and ***96*** may implement a priority scheme other than, or in addition to, this first-in-first-out scheme. For example, suppose the thread $\mathbf{100_1}$ publishes first data, and subsequently the thread $\mathbf{100_2}$ publishes second data but

also publishes to the output queue object ***96*** a priority flag associated with the second data. Because the second data has priority over the first data, the output reader object ***92*** notifies the data-transfer object ***86_{2b}*** of the published second data in the buffer ***106_2*** before notifying the data-transfer object ***86_{1b}*** of the published first data in the buffer ***106_1***.

[69] Then, the data-transfer object ***86_{1b}*** retrieves the published data from the buffer ***106_1*** and formats the data in a predetermined manner. For example, the object ***86_{1b}*** generates a message that includes the published data (*i.e.*, the payload) and a header that, *e.g.*, identifies the destination of the data within the accelerator ***44***. This message may have an industry-standard format such as the Rapid IO (input/output) format. Because the generation of such a message is conventional, it is not discussed further.

[70] After the data-transfer object ***86_{1b}*** formats the published data, it sends the formatted data to the communication object ***88***.

[71] Next, the communication object ***88*** sends the formatted data to the pipeline accelerator ***44*** via the bus ***50***. The communication object ***88*** is designed to implement the communication protocol (*e.g.*, Rapid IO, TCP/IP) used to transfer data between the host processor ***42*** and the accelerator ***44***. For example, the communication object ***88*** implements the required hand shaking and other transfer parameters (*e.g.*, arbitrating the sending and receiving of messages on the bus ***50***) that the protocol requires. Alternatively, the data-transfer object ***86_{xb}*** can implement the communication protocol, and the communication object ***88*** can be omitted. However, this latter alternative is less efficient because it requires all the data-transfer objects ***86_{xb}*** to include additional code and functionality.

[72] The pipeline accelerator ***44*** then receives the formatted data, recovers the data from the message (*e.g.*, separates the data from the header if there is a header), directs the data to the proper destination within the accelerator, and processes the data.

[73] Still referring to **FIGS. 3-5**, an example of the pipeline accelerator ***44*** (**FIG. 3**) sending data to the host processor ***42*** (**FIG. 3**) is discussed in conjunction with the channel ***104_2***.

[74] First, the pipeline accelerator ***44*** generates and formats data. For example, the accelerator ***44*** generates a message that includes the data payload and a header that, *e.g.*, identifies the destination threads ***100_1*** and ***100_2***, which are the threads that are to receive and process the data. As discussed above, this message may have an industry-standard format such as the Rapid IO (input/output) format.

[75] Next, the accelerator ***44*** drives the formatted data onto the bus ***50*** in a conventional manner.

[76] Then, the communication object ***88*** receives the formatted data from the bus ***50*** and provides the formatted data to the data-transfer object ***86_{2b}***. In one embodiment, the formatted data is in the form of a message, and the communication object ***88*** analyzes the message header (which, as discussed above, identifies the destination threads ***100_1*** and ***100_2***) and provides the message to the data-transfer object ***86_{2b}*** in response to the header. In another embodiment, the communication object ***88*** provides the message to all of the data-transfer objects ***86_{nb}***, each of which analyzes the message header and processes the message only if its function is to provide data to the destination threads ***100_1*** and ***100_2***. Consequently, in this example, only the data-transfer object ***86_{2b}*** processes the message.

[77] Next, the data-transfer object ***86_{2b}*** loads the data received from the communication object ***88*** into the buffer ***106_2***. For example, if the data is contained within a message payload, the data-transfer object ***86_{2b}*** recovers the data from the message (*e.g.*, by stripping the header) and loads the recovered data into the buffer ***106_2***.

[78] Then, the data-transfer object ***86_{2a}*** determines that the buffer ***106_2*** has received new data from the data-transfer object ***86_{2b}***. The input reader object ***90*** may periodically instruct the data-transfer object ***86_{2a}*** to check the buffer ***106_2*** for newly received data. Alternatively, the input reader object ***90*** notifies the data-transfer object ***86_{2a}*** when the buffer ***106_2*** has received newly published data. Specifically, the input queue object ***94*** generates and stores a unique identifier (not shown) in response to the data-transfer object ***86_{2b}*** storing the published data in the buffer ***106_2***. In response to this identifier, the input reader object ***90*** notifies the data-transfer object ***86_{2a}*** that the buffer ***106_2*** contains newly published data. As

discussed above in conjunction with the output reader and output queue objects ***92*** and ***96***, where multiple buffers ***106*** contain respective newly published data, then the input queue object ***94*** may record the order in which this data was published, and the input reader object ***90*** may notify the respective data-transfer objects ***86***$_{xa}$ in the same order. Alternatively, where multiple buffers ***106*** contain respective newly published data, the input reader and input queue objects ***90*** and ***94*** may implement a priority scheme other than, or in addition to, this first-in-first-out scheme.

[79] Next, the data-object ***86***$_{2a}$ transfers the data from the buffer ***106***$_2$ to the subscriber threads ***100***$_1$ and ***100***$_2$, which perform respective operations on the data.

[80] Referring to **FIG. 5**, an example of one thread receiving and processing data from another thread is discussed in conjunction with the thread ***100***$_4$ receiving and processing data published by the thread ***100***$_3$.

[81] In one embodiment, the thread ***100***$_3$ publishes the data directly to the thread ***100***$_4$ via the optional connection (dashed line) ***102***.

[82] In another embodiment, the thread ***100***$_3$ publishes the data to the thread ***100***$_4$ via the channels ***104***$_5$ and ***104***$_6$. Specifically, the data-transfer object ***86***$_{5a}$ loads the published data into the buffer ***106***$_5$. Next, the data-transfer object ***86***$_{5b}$ retrieves the data from the buffer ***106***$_5$ and transfers the data to the communication object ***88***, which publishes the data to the data-transfer object ***86***$_{6b}$. Then, the data-transfer object ***86***$_{6b}$ loads the data into the buffer ***106***$_6$. Next, the data-transfer object ***86***$_{6a}$ transfers the data from the buffer ***106***$_6$ to the thread ***100***$_4$. Alternatively, because the data is not being transferred via the bus ***50***, then one may modify the data-transfer object ***86***$_{5b}$ such that it loads the data directly into the buffer ***106***$_6$, thus bypassing the communication object ***88*** and the data-transfer object ***86***$_{6b}$. But modifying the data-transfer object ***86***$_{5b}$ to be different from the other data-transfer objects ***86*** may increase the complexity modularity of the message handler ***64***.

[83] Still referring to **FIG. 5**, additional data-transfer techniques are contemplated. For example a single thread may publish data to multiple locations within the pipeline accelerator ***44*** (**FIG. 3**) via respective multiple channels. Alternatively, as discussed in previously cited U.S. Patent App. Serial Nos. 10/684,102 entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED

SYSTEM AND METHOD and 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD, the accelerator ***44*** may receive data via a single channel ***104*** and provide it to multiple locations within the accelerator. Furthermore, multiple threads (*e.g.*, threads ***100_1*** and ***100_2***) may subscribe to data from the same channel (*e.g.*, channel ***104_2***). In addition, multiple threads (*e.g.*, threads ***100_2*** and ***100_3***) may publish data to the same location within the accelerator ***44*** via the same channel (*e.g.*, channel ***104_3***), although the threads may publish data to the same accelerator location via respective channels ***104***.

[84] **FIG. 6** is a functional block diagram of the exception manager ***82***, the data-transfer objects ***86***, and the interface memory ***48*** according to an embodiment of the invention.

[85] The exception manager ***82*** receives and logs exceptions that may occur during the initialization or operation of the pipeline accelerator ***44*** (**FIG. 3**). Generally, an exception is a designer-defined event where the accelerator ***44*** acts in an undesired manner. For example, a buffer (not shown) that overflows may be an exception, and thus cause the accelerator ***44*** to generate an exception message and send it to the exception manager ***82***. Generation of an exception message is discussed in previously cited U.S. Patent App. Serial No. 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD.

[86] The exception manager ***82*** may also handle exceptions that occur during the initialization or operation of the pipeline accelerator ***44*** (**FIG. 3**). For example, if the accelerator ***44*** includes a buffer (not shown) that overflows, then the exception manager ***82*** may cause the accelerator to increase the size of the buffer to prevent future overflow. Or, if a section of the accelerator ***44*** malfunctions, the exception manager ***82*** may cause another section of the accelerator or the data-processing application ***80*** to perform the operation that the malfunctioning section was intended to perform. Such exception handling is further discussed below and in previously cited U.S. Patent App. Serial No. 10/;683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD.

[87] To log and/or handle accelerator exceptions, the exception manager ***82*** subscribes to data from one or more subscriber threads ***100*** (**FIG. 5**) and determines from this data whether an exception has occurred.

[88] In one alternative, the exception manager ***82*** subscribes to the same data as the subscriber threads ***100*** (**FIG. 5**) subscribe to. Specifically, the manager ***82*** receives this data via the same respective channels $\boldsymbol{104_s}$ (which include, *e.g.*, channel $\boldsymbol{104_2}$ of **FIG. 5**) from which the subscriber threads ***100*** (which include, *e.g.*, threads $\boldsymbol{100_1}$ and $\boldsymbol{100_2}$ of **FIG. 5**) receive the data. Consequently, the channels $\boldsymbol{104_s}$ provide this data to the exception manager ***82*** in the same manner that they provide this data to the subscriber threads ***100***.

[89] In another alternative, the exception manager ***82*** subscribes to data from dedicated channels ***106*** (not shown), which may receive data from sections of the accelerator ***44*** (**FIG. 3**) that do not provide data to the threads ***100*** via the subscriber channels $\boldsymbol{104_s}$. Where such dedicated channels ***104*** are used, the object factory ***98*** (**FIG. 4**) generates the data-transfer objects ***86*** for these channels during initialization of the host processor ***42*** as discussed above in conjunction with **FIG. 4**. The exception manager ***82*** may subscribe to the dedicated channels ***106*** exclusively or in addition to the subscriber channels $\boldsymbol{104_s}$.

[90] To determine whether an exception has occurred, the exception manager ***82*** compares the data to exception codes stored in a registry (not shown) within the memory ***66*** (**FIG. 3**). If the data matches one of the codes, then the exception manager ***82*** determines that the exception corresponding to the matched code has occurred.

[91] In another alternative, the exception manager ***82*** analyzes the data to determine if an exception has occurred. For example, the data may represent the result of an operation performed by the accelerator ***44***. The exception manager ***82*** determines whether the data contains an error, and, if so, determines that an exception has occurred and the identity of the exception.

[92] After determining that an exception has occurred, the exception manager ***82*** logs, *e.g.*, the corresponding exception code and the time of occurrence, for later use such as during a debug of the accelerator ***44***. The

exception manager ***82*** may also determine and convey the identity of the exception to, *e.g.*, the system designer, in a conventional manner.

[93] Alternatively, in addition to logging the exception, the exception manager ***82*** may implement an appropriate procedure for handling the exception. For example, the exception manager ***82*** may handle the exception by sending an exception-handling instruction to the accelerator ***44***, the data-processing application ***80***, or the configuration manager ***84***. The exception manager ***82*** may send the exception-handling instruction to the accelerator ***44*** either via the same respective channels $\boldsymbol{104_p}$ (*e.g.*, channel $\boldsymbol{104_1}$ of **FIG. 5**) through which the publisher threads ***100*** (*e.g.*, thread $\boldsymbol{100_1}$ of **FIG. 5**) publish data, or through dedicated exception-handling channels ***104*** (not shown) that operate as described above in conjunction with **FIG. 5**. If the exception manager ***82*** sends instructions via other channels ***104***, then the object factory ***98*** (**FIG. 4**) generates the data-transfer objects ***86*** for these channels during initialization of the host processor ***42*** as described above in conjunction with **FIG. 4**. The exception manager ***82*** may publish exception-handling instructions to the data-processing application ***80*** and to the configuration manager ***84*** either directly (as indicated by the dashed lines ***85*** and ***89*** in **FIG. 4**) or via the channels $\boldsymbol{104_{dpa1}}$ and $\boldsymbol{104_{dpa2}}$ (application ***80***) and channels $\boldsymbol{104_{cm1}}$ and $\boldsymbol{104_{cm2}}$ (configuration manager ***84***), which the object factory ***98*** also generates during the initialization of the host processor ***42***.

[94] Still referring to **FIG. 6**, as discussed below the exception-handling instructions may cause the accelerator ***44***, data-processing application ***80***, or configuration manager ***84*** to handle the corresponding exception in a variety of ways.

[95] When sent to the accelerator ***44***, the exception-handling instruction may change the soft configuration or the functioning of the accelerator. For example, as discussed above, if the exception is a buffer overflow, the instruction may change the accelerator's soft configuration (*i.e.*, by changing the contents of a soft configuration register) to increase the size of the buffer. Or, if a section of the accelerator ***44*** that performs a particular operation is malfunctioning, the instruction may change the accelerator's functioning by causing the accelerator to take the disabled section "off line." In this latter case, the exception manager ***82*** may, via

additional instructions, cause another section of the accelerator ***44***, or the data-processing application ***80***, to "take over" the operation from the disabled accelerator section as discussed below. Altering the soft configuration of the accelerator ***44*** is further discussed in previously cited U.S. Patent App. Serial No. 10/683,929 entitled PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-13-3).

[96] When sent to the data-processing application ***80***, the exception-handling instructions may cause the data-processing application to "take over" the operation of a disabled section of the accelerator ***44*** that has been taken off line. Although the processing unit ***62*** (**FIG. 3**) may perform this operation more slowly and less efficiently than the accelerator ***44***, this may be preferable to not performing the operation at all. This ability to shift the performance of an operation from the accelerator ***44*** to the processing unit ***62*** increases the flexibility, reliability, maintainability, and fault-tolerance of the peer-vector machine ***40*** (**FIG. 3**).

[97] And when sent to the configuration manager ***84***, the exception-handling instruction may cause the configuration manager to change the hard configuration of the accelerator ***44*** so that the accelerator can continue to perform the operation of a malfunctioning section that has been taken off line. For example, if the accelerator ***44*** has an unused section, then the configuration manager ***84*** may configure this unused section to perform the operation that was to be the malfunctioning section. If the accelerator ***44*** has no unused section, then the configuration manager ***84*** may reconfigure a section of the accelerator that currently performs a first operation to perform a second operation of, *i.e.*, take over for, the malfunctioning section. This technique may be useful where the first operation can be omitted but the second operation cannot, or where the data-processing application ***80*** is more suited to perform the first operation than it is the second operation. This ability to shift the performance of an operation from one section of the accelerator ***44*** to another section of the accelerator increases the flexibility, reliability, maintainability, and fault-tolerance of the peer-vector machine ***40*** (**FIG. 3**).

[98] Referring to **FIG. 7**, the configuration manager ***84*** loads the firmware that defines the hard configuration of the accelerator ***44*** during initialization of the

peer-vector machine ***40*** (**FIG. 3**), and, as discussed above in conjunction with **FIG. 6**, may load firmware that redefines the hard configuration of the accelerator in response to an exception according to an embodiment of the invention. As discussed below, the configuration manager ***84*** often reduces the complexity of designing and modifying the accelerator ***44*** and increases the fault-tolerance, reliability, maintainability, and flexibility of the peer-vector machine ***40*** (**FIG. 3**).

[99] During initialization of the peer-vector machine ***40***, the configuration manager ***84*** receives configuration data from the accelerator configuration registry ***70***, and loads configuration firmware identified by the configuration data. The configuration data are effectively instructions to the configuration manager ***84*** for loading the firmware. For example, if a section of the initialized accelerator ***44*** performs an FFT, then one designs the configuration data so that the firmware loaded by the manager ***84*** implements an FFT in this section of the accelerator. . Consequently, one can modify the hard configuration of the accelerator ***44*** by merely generating or modifying the configuration data before initialization of the peer-vector machine ***40***. Because generating and modifying the configuration data is often easier than generating and modifying the firmware directly — particularly if the configuration data can instruct the configuration manager ***84*** to load existing firmware from a library — the configuration manager ***84*** typically reduces the complexity of designing and modifying the accelerator ***44***.

[100] Before the configuration manager ***84*** loads the firmware identified by the configuration data, the configuration manager determines whether the accelerator ***44*** can support the configuration defined by the configuration data. For example, if the configuration data instructs the configuration manager ***84*** to load firmware for a particular PLIC (not shown) of the accelerator ***44***, then the configuration manager ***84*** confirms that the PLIC is present before loading the data. If the PLIC is not present, then the configuration manager ***84*** halts the initialization of the accelerator ***44*** and notifies an operator that the accelerator does not support the configuration.

[101] After the configuration manager ***84*** confirms that the accelerator supports the defined configuration, the configuration manager loads the firmware into the accelerator ***44***, which sets its hard configuration with the firmware, *e.g.*, by

loading the firmware into the firmware memory ***52***. Typically, the configuration manager ***84*** sends the firmware to the accelerator ***44*** via one or more channels ***104***$_t$ that are similar in generation, structure, and operation to the channels ***104*** of **FIG. 5**. The configuration manager ***84*** may also receive data from the accelerator ***44*** via one or more channels ***104***$_u$. For example, the accelerator ***44*** may send confirmation of the successful setting of its hard configuration to the configuration manager ***84***.

[102] After the hard configuration of the accelerator ***44*** is set, the configuration manager ***84*** may set the accelerator's hard configuration in response to an exception-handling instruction from the exception manager ***84*** as discussed above in conjunction with **FIG. 6**. In response to the exception-handling instruction, the configuration manager ***84*** downloads the appropriate configuration data from the registry ***70***, loads reconfiguration firmware identified by the configuration data, and sends the firmware to the accelerator ***44*** via the channels ***104***$_t$. The configuration manager ***84*** may receive confirmation of successful reconfiguration from the accelerator ***44*** via the channels ***104***$_u$. As discussed above in conjunction with **FIG. 6**, the configuration manager ***84*** may receive the exception-handling instruction directly from the exception manager ***82*** via the line ***89*** (**FIG. 4**) or indirectly via the channels ***104***$_{cm1}$ and ***104***$_{cm2}$.

[103] The configuration manager ***84*** may also reconfigure the data-processing application ***80*** in response to an exception-handling instruction from the exception manager ***84*** as discussed above in conjunction with **FIG. 6**. In response to the exception-handling instruction, the configuration manager ***84*** instructs the data-processing application ***80*** to reconfigure itself to perform an operation that, due to malfunction or other reason, the accelerator ***44*** cannot perform. The configuration manager ***84*** may so instruct the data-processing application ***80*** directly via the line ***87*** (**FIG. 4**) or indirectly via channels ***104***$_{dp1}$ and ***104***$_{dp2}$, and may receive information from the data-processing application, such as confirmation of successful reconfiguration, directly or via another channel ***104*** (not shown). Alternatively, the exception manager ***82*** may send an exception-handling instruction to the data-processing ***80***, which reconfigures itself, thus bypassing the configuration manager ***82***.

[104] Still referring to **FIG. 7**, alternate embodiments of the configuration manager ***82*** are contemplated. For example, the configuration manager ***82*** may reconfigure the accelerator ***44*** or the data-processing application ***80*** for reasons other than the occurrence of an accelerator malfunction.

[105] The preceding discussion is presented to enable a person skilled in the art to make and use the invention. Various modifications to the embodiments will be readily apparent to those skilled in the art, and the generic principles herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

WHAT IS CLAIMED IS:

1. A computing machine, comprising:

a first buffer;

a processor coupled to the buffer and operable to,

execute an application, a first data-transfer object, and a second data-transfer object,

publish data under the control of the application,

load the published data into the buffer under the control of the first data-transfer object, and

retrieve the published data from the buffer under the control of the second data-transfer object.

2. The computing machine of claim 1 wherein the first and second data-transfer objects respectively comprise first and second instances of the same object code.

3. The computing machine of claim 1 wherein the processor comprises:

a processing unit operable to execute the application and publish the data under the control of the application; and

a data-transfer handler operable to execute the first and second data-transfer objects, to load the published data into the buffer under the control of the first data-transfer object, and to retrieve the published data under the control of the second data-transfer object.

4. The computing machine of claim 1 wherein the processor is further operable to execute a thread of the application and to publish the data under the control of the thread.

5. The computing machine of claim 1 wherein the processor is further operable to:

execute a queue object and a reader object;

store a queue value under the control of the queue object, the queue value reflecting the loading of the published data into the buffer;

read the queue value under the control of the reader object;

notify the second software object that the published data occupies the buffer under the control of the reader object and in response to the queue value; and

retrieve the published data from the storage location under the control of the second data-transfer object and in response to the notification.

6. The computing machine of claim 1, further comprising:

a bus; and

wherein the processor is operable to execute an communication object and to drive the retrieved data onto the bus under the control of the communication object.

7. The computing machine of claim 1, further comprising:

a second buffer; and

wherein the processor is operable to provide the retrieved data to the second buffer under the control of the second data-transfer object.

8. The computing machine of claim 1 wherein the processor is further operable to generate a message that includes a header and the retrieved data under the control of the second data-transfer object.

9. The computing machine of claim 1 wherein:

the first and second data-transfer objects respectively comprise first and second instances of the same object code; and

the processor is operable to execute an object factory and to generate the object code under the control of the object factory.

10. A computing machine, comprising:

a first buffer;

a processor coupled to the buffer and operable to,

execute first and second data-transfer objects and an application,

retrieve data and load the retrieved data into the buffer under the control of the first data-transfer object,

unload the data from the buffer under the control of the second data-transfer object, and

process the unloaded data under the control of the application.

11. The computing machine of claim 10 wherein the first and second data-transfer objects respectively comprise first and second instances of the same object code.

12. The computing machine of claim 10 wherein the processor comprises:

a processing unit operable to execute the application and process the unloaded data under the control of the application; and

a data-transfer handler operable to execute the first and second data-transfer objects, to retrieve the data from the bus and load the data into the buffer under the control of the first data-transfer object, and to unload the data from the buffer under the control of the second data-transfer object.

13. The computing machine of claim 10 wherein the processor is further operable to execute a thread of the application and to process the unloaded data under the control of the thread.

14. The computing machine of claim 10 wherein the processor is further operable to:

execute a queue object and a reader object;

store a queue value under the control of the queue object, the queue value reflecting the loading of the published data into the first buffer;

read the queue value under the control of the reader object;

notify the second data-transfer object that the published data occupies the buffer under the control of the reader object and in response to the queue value; and

unload the published data from the buffer under the control of the second data-transfer object and in response to the notification.

15. The computing machine of claim 10, further comprising:

a second buffer; and

wherein the processor is operable to retrieve the data from the second buffer under the control of the first data-transfer object.

16. The computing machine of claim 10, further comprising:

a bus; and

wherein the processor is operable to execute an communication object, to receive the data from the bus under the control of the communication object, and to retrieve the data from the communication object under the control of the first data-transfer object.

17. The computing machine of claim 10 wherein:

the first and second data-transfer objects respectively comprise first and second instances of the same object code; and

the processor is operable to execute an object factory and to generate the object code under the control of the object factory.

18. The computing machine of claim 10 wherein the processor is further operable to recover the data from a message that includes a header and the data under the control of the first data-transfer object.

19. A peer-vector machine, comprising:

a buffer;

a bus;

a processor coupled to the buffer and to the bus and operable to,

execute an application, first and second data-transfer objects, and an communication object,

publish data under the control of the application,

oad the published data into the buffer under the control of the first data-transfer object,

retrieve the published data from the buffer under the control of the second data-transfer object, and

drive the published data onto the bus under the control of the communication object; and

a pipeline accelerator coupled to the bus and operable to receive the published data from the bus and to process the received published data.

20. The peer-vector machine of claim 19 wherein:

the processor is further operable to construct a message that includes the published data under the control of the second data-transfer object and to drive the message onto the bus under the control of the communication object; and

the pipeline accelerator is operable to receive the message from the bus and to recover the published data from the message.

21. The peer-vector machine of claim 19, further comprising:

a registry coupled to the host processor and operable to store object data; and

wherein the processor is operable to,

execute an object factory, and

to generate the first and second data-transfer objects and the communication object from the object data under the control of the object factory.

22. A peer-vector machine, comprising:

a buffer;

a bus;

a pipeline accelerator coupled to the bus and operable to generate data and to drive the data onto the bus; and

a processor coupled to the buffer and to the bus and operable to,

execute an application, first and second data-transfer objects, and an communication object,

receive the data from the bus under the control of the communication object,

load the received data into the buffer under the control of the first data-transfer object,

unload the data from the buffer under the control of the second data-transfer object, and

process the unloaded data under the control of the application.

23. The peer-vector machine of claim 22 wherein:

the pipeline accelerator is further operable to construct a message that includes the data and to drive the message onto the bus; and

the processor is operable to,

receive the message from the bus under the control of the communication object, and

recover the data from the message under the control of the first data-transfer object.

24. The peer-vector machine of claim 22, further comprising:

a registry coupled to the host processor and operable to store object data; and wherein the processor is operable to,

execute an object factory, and

to generate the first and second data-transfer objects and the communication object from the object data under the control of the object factory.

25. A peer-vector machine, comprising:

a first buffer;

a bus;

a processor coupled to the buffer and to the bus and operable to,

execute a configuration manager, first and second data-transfer objects, and a communication object,

load configuration firmware into the buffer under the control of the configuration manager and the first data-transfer object,

retrieve the configuration firmware from the buffer under the control of the second data-transfer object, and

drive the configuration firmware onto the bus under the control of the communication object; and

a pipeline accelerator coupled to the bus and operable to receive the configuration firmware and to configure itself with the configuration firmware.

26. The peer-vector machine of claim 25 wherein:

the processor is further operable to construct a message that includes the configuration firmware under the control of the second data-transfer object and to drive the message onto the bus under the control of the communication object; and

the pipeline accelerator is operable to receive the message from the bus and to recover the configuration firmware from the message.

27. The peer-vector machine of claim 25, further comprising:

a registry coupled to the processor and operable to store configuration data; and

wherein the processor is operable to locate the configuration firmware from the configuration data under the control of the configuration manager.

28. The peer-vector machine of claim 25, further comprising:

a second buffer; and

wherein the processor is operable to:

execute an application and third and fourth data-transfer objects,

generate a configuration instruction under the control of the configuration manager,

load the configuration instruction into the second buffer under the control of the third data-transfer object,

retrieve the configuration instruction from the second buffer under the control of the fourth data-transfer object, and

configure the application to perform an operation corresponding to the configuration instruction under the control of the application.

29. The peer-vector machine of claim 25 wherein the processor is operable to:

generate a configuration instruction under the control of the configuration manager; and

configure the application to perform an operation corresponding to the configuration instruction under the control of the application.

30. The peer-vector machine of claim 25 wherein the configuration manager is operable to confirm that the pipeline accelerator supports a configuration defined by the configuration data before loading the firmware.

31. A peer-vector machine, comprising:

a first buffer;

a bus;

a pipeline accelerator coupled to the bus and operable to generate exception data and to drive the exception data onto the bus; and

a processor coupled to the buffer and to the bus and operable to,

execute an exception manager, first and second data-transfer objects, and an communication object,

receive the exception data from the bus under the control of the communication object,

load the received exception data into the buffer under the control of the first data-transfer object,

unload the exception data from the buffer under the control of the second data-transfer object, and

process the unloaded exception data under the control of the exception manager.

32. The peer-vector machine of claim 31 wherein:

the pipeline is further operable to construct a message that includes the exception data and to drive the message onto the bus; and

the processor is operable to receive the message from the bus under the control of the communication object and to recover the exception data from the message under the control of the first data-transfer object.

33. The peer-vector machine of claim 31, further comprising:

a second buffer;

wherein the processor is further operable to,

execute a configuration manager and third and fourth data-transfer objects,

generate configuration firmware under the control of the configuration manager in response to the exception data,

load the configuration firmware into the second buffer under the control of the third data-transfer object,

unload the configuration instruction from the second buffer under the control of the fourth data-transfer object, and

drive the configuration firmware onto the bus under the control of the communication object; and

wherein the pipeline accelerator is operable to receive the configuration firmware from the bus and reconfigure itself with the firmware.

34. The peer-vector machine of claim 31 wherein the processor is further operable to:

execute an application and a configuration manager;

generate a configuration instruction under the control of the configuration manager in response to the exception data; and

reconfigure the application under the control of the application in response to the configuration instruction.

35. A peer-vector machine, comprising:

a configuration registry operable to store configuration data;

a processor coupled to the configuration registry and operable to locate configuration firmware from the configuration data; and

a pipeline accelerator coupled to the processor and operable to configure itself with the configuration firmware.

36. A peer-vector machine, comprising:

a configuration registry operable to store configuration data;

a pipeline accelerator; and

a processor coupled to the configuration registry and to the pipeline accelerator and operable to retrieve configuration firmware in response to the configuration data and to configure the pipeline accelerator with the configuration firmware.

37. A method, comprising:

publishing data with an application;

loading the published data into a first buffer with a first data-transfer object; and

retrieving the published data from the buffer with a second data-transfer object.

38. The method of claim 37 wherein publishing the data comprises publishing the data with a thread of the application.

39. The method of claim 37, further comprising:

generating a queue value that corresponds to the presence of the published data in the buffer;

notifying the second data-transfer object that the published data occupies the buffer in response to the queue value; and

wherein retrieving the published data comprises retrieving the published data from the storage location with the second data-transfer object in response to the notification.

40. The method of claim 37, further comprising driving the retrieved data onto a bus with a communication object.

41. The method of claim 37, further comprising loading the retrieved data into a second buffer with the second data-transfer object.

42. The method of claim 37, further comprising:

generating a header for the retrieved data with the second data-transfer object; and

combining the header and the retrieved data into a message with the second data-transfer object.

43. The method of claim 37, further comprising:

generating data-transfer object code with an object factory;

generating the first data-transfer object as a first instance of the object code; and

generating the second data-transfer object as a second instance of the object code.

44. The method of claim 37, further comprising receiving and processing the data from the second data-transfer object with a pipeline accelerator.

45. A method, comprising:

retrieving data and loading the retrieved data into a first buffer with a first data-transfer object,

unloading the data from the buffer with a second data-transfer object; and

processing the unloaded data with an application.

46. The method of claim 45 wherein processing the unloaded data comprises processing the unloaded data with a thread of the application.

47. The method of claim 45, further comprising:

generating a queue value that corresponds to the presence of the data in the buffer;

notifying the second data-transfer object that the data occupies the buffer in response to the queue value; and

wherein unloading the data comprises unloading the data from the buffer with the first data-transfer object in response to the notification.

48. The method of claim 45 wherein retrieving the data comprises retrieving the data from a second buffer with the first data-transfer object.

49. The method of claim 45, further comprising:

receiving the data from a bus with an communication object; and

wherein retrieving the data comprises retrieving the data from the communication object under with the first data-transfer object.

50. The method of claim 45, further comprising providing the data to the first data-transfer object with a pipeline accelerator.

51. A method, comprising:

publishing data with an application running on a processor;

loading the published data into a buffer with a first data-transfer object running on the processor;

retrieving the published data from the buffer with a second data-transfer object running on the processor;

driving the retrieved published data onto a bus with an communication object running on the processor; and

receiving the published data from the bus and processing the published data with a pipeline accelerator.

52. The method of claim 51, further comprising:

generating a message that includes a header and the published data with the second data-transfer object;

wherein driving the data onto the bus comprises driving the message onto the bus with the communication object; and

receiving and processing the published data comprises receiving the message and recovering the published data from the message with the pipeline accelerator.

53. A method, comprising:

generating data and driving the data onto a bus with a pipeline accelerator;

receiving the data from the bus with the communication object;

loading the received data into a buffer under with a first data-transfer object;

unloading the data from the buffer with a second data-transfer object; and

processing the unloaded data with an application.

54. The method of claim 53, further comprising:

wherein generating the data comprises constructing a message that includes a header and the data with the pipeline accelerator;

wherein driving the data comprises driving the message onto the bus with the pipeline accelerator;

wherein receiving the data comprises receiving the message from the bus with the communication object; and

recovering the data from the message with the first data-transfer object.

55. A method, comprising:

retrieving configuration firmware with a configuration manager;

loading the configuration firmware into a first buffer with a first communication object;

retrieving the configuration firmware from the buffer with a second communication object;

driving the configuration firmware onto a bus with an communication object;

receiving the configuration firmware with a pipeline accelerator; and

configuring the pipeline accelerator with the configuration firmware.

56. The method of claim 55, further comprising:

generating a configuration instruction with the configuration manager; and

configuring the application to perform an operation corresponding to the configuration instruction.

57. The method of claim 55, further comprising:

generating a configuration instruction with the configuration manager;

loading the configuration instruction into a second buffer with a third communication object;

retrieving the configuration instruction from the second buffer with a fourth communication object; and

configuring the application to perform an operation corresponding to the configuration instruction.

58. A method, comprising:

generating exception data and driving the exception data onto a bus with a pipeline accelerator;

receiving the exception data from the bus with a communication object;

loading the received exception data into a buffer with a first data-transfer object;

unloading the exception data from the buffer with a second data-transfer object; and

processing the unloaded exception data under with an exception manager.

59. The method of claim 58, further comprising:

retrieving configuration firmware with a configuration manager in response to the exception data,

loading the configuration firmware into a second buffer with a third transfer object;

unloading the configuration instruction from the second buffer with a fourth data-transfer object;

driving the configuration firmware onto the bus with the communication object; and

reconfiguring the pipeline accelerator with the configuration firmware.

60. The method of claim 58, further comprising:

generating a configuration instruction with a configuration manager in response to the error data; and

reconfiguring the application in response to the configuration instruction.

61. A method, comprising:

retrieving configuration firmware pointed to by configuration data stored in a configuration registry during an initialization of a computing machine; and

configuring a pipeline accelerator of the computing machine with the configuration firmware.

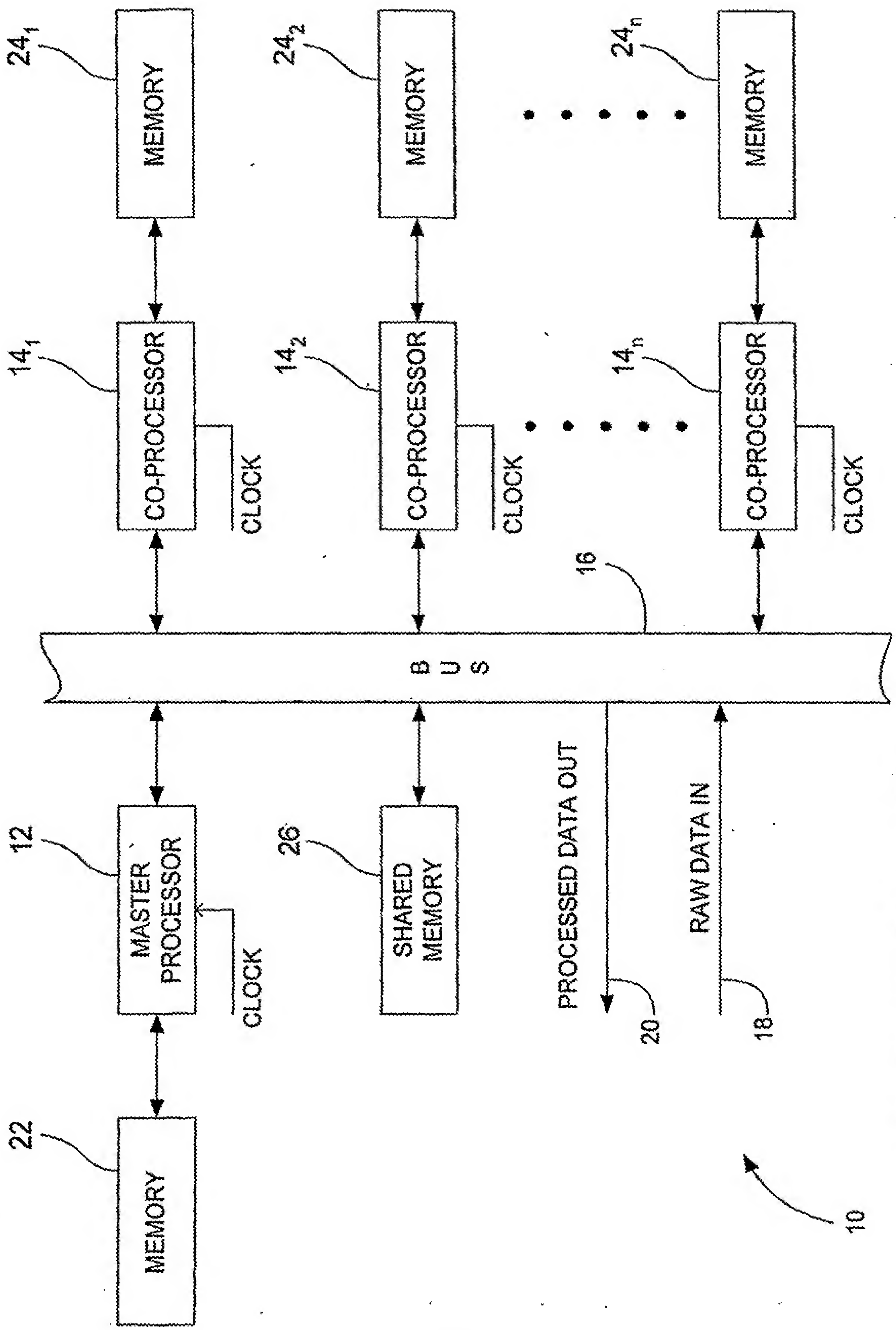


FIG. 1 (PRIOR ART)

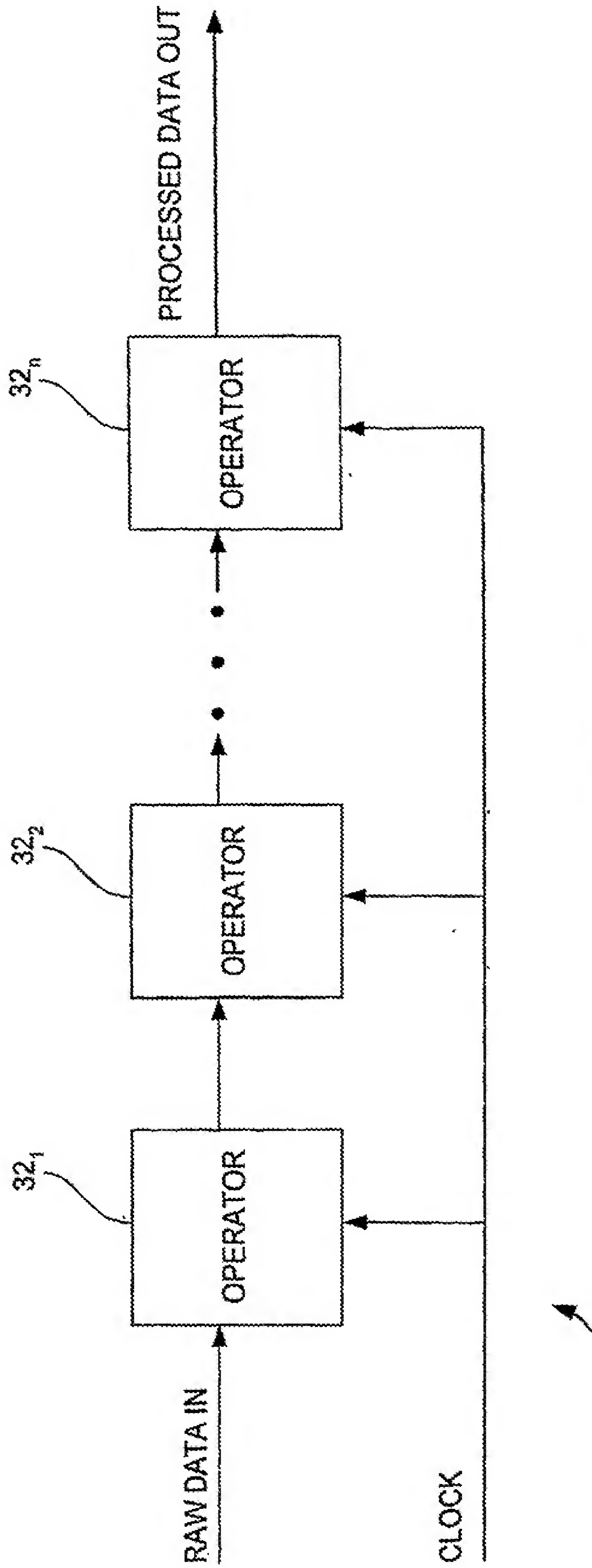
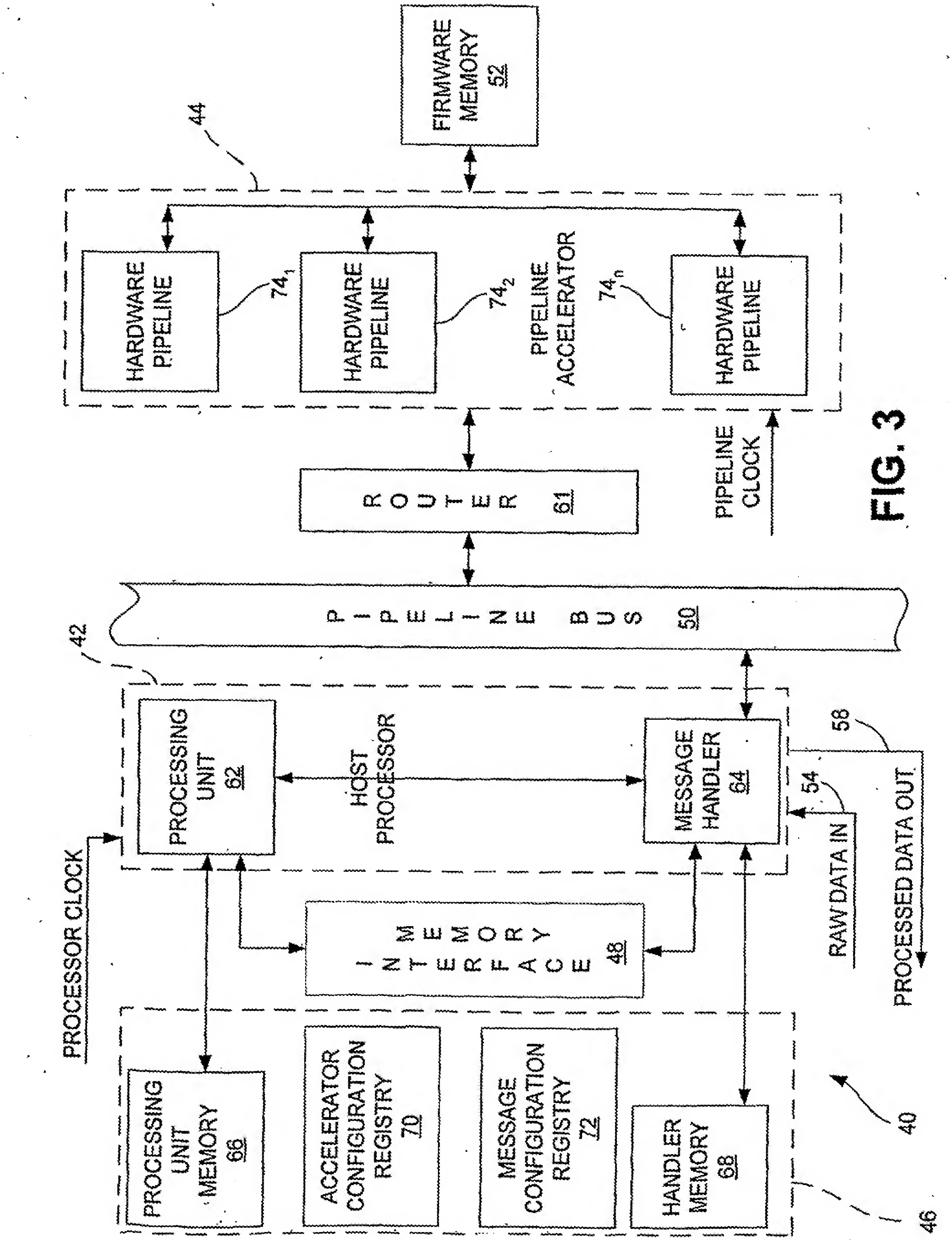

PROCESSED DATA OUT
32_n
OPERATOR
32_2
OPERATOR
32_1
OPERATOR
RAW DATA IN
CLOCK
30

FIG. 2



FIRMWARE MEMORY 52
44
HARDWARE PIPELINE
74_1
HARDWARE PIPELINE
74_2
PIPELINE ACCELERATOR
74_n
HARDWARE PIPELINE
PIPELINE CLOCK
ROUTER 61
PIPELINE BUS 50
42
PROCESSING UNIT 62
HOST PROCESSOR
MESSAGE HANDLER 64
58
54
RAW DATA IN
PROCESSED DATA OUT
PROCESSOR CLOCK
MEMORY INTERFACE 48
PROCESSING UNIT MEMORY 66
ACCELERATOR CONFIGURATION REGISTRY 70
MESSAGE CONFIGURATION REGISTRY 72
HANDLER MEMORY 68
40
46

FIG. 3

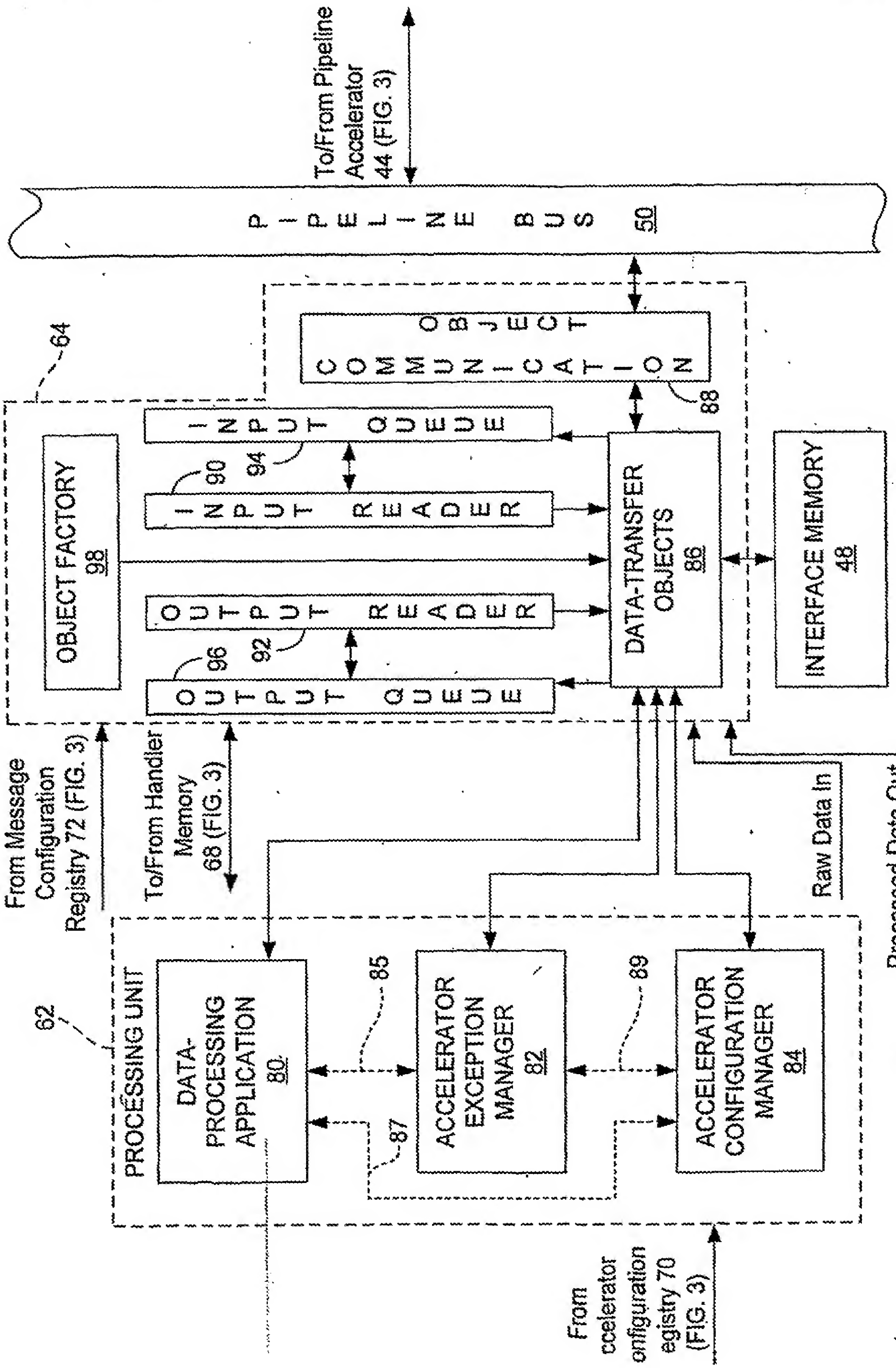


FIG. 4

SUBSTITUTE SHEET (RULE 26)

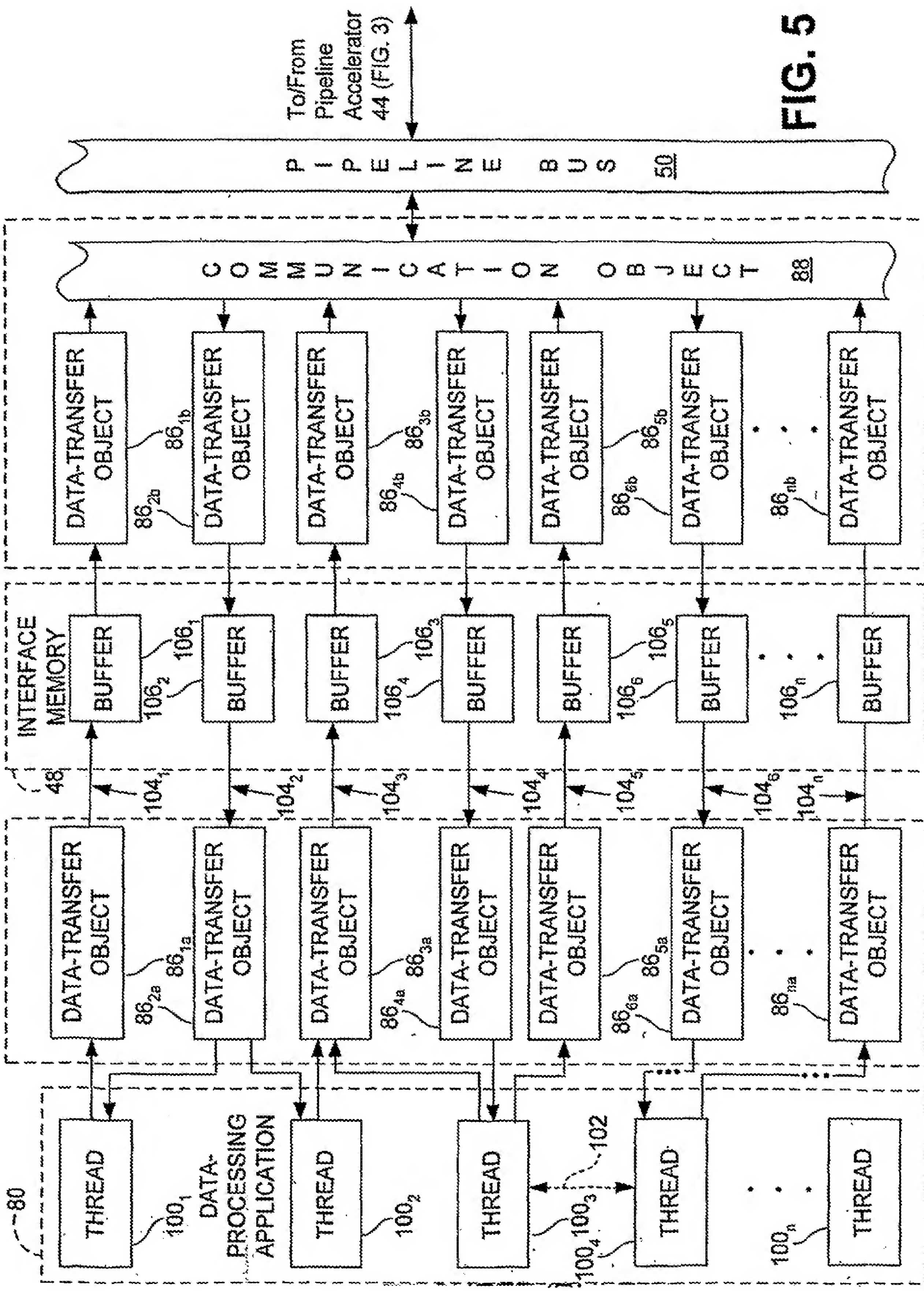
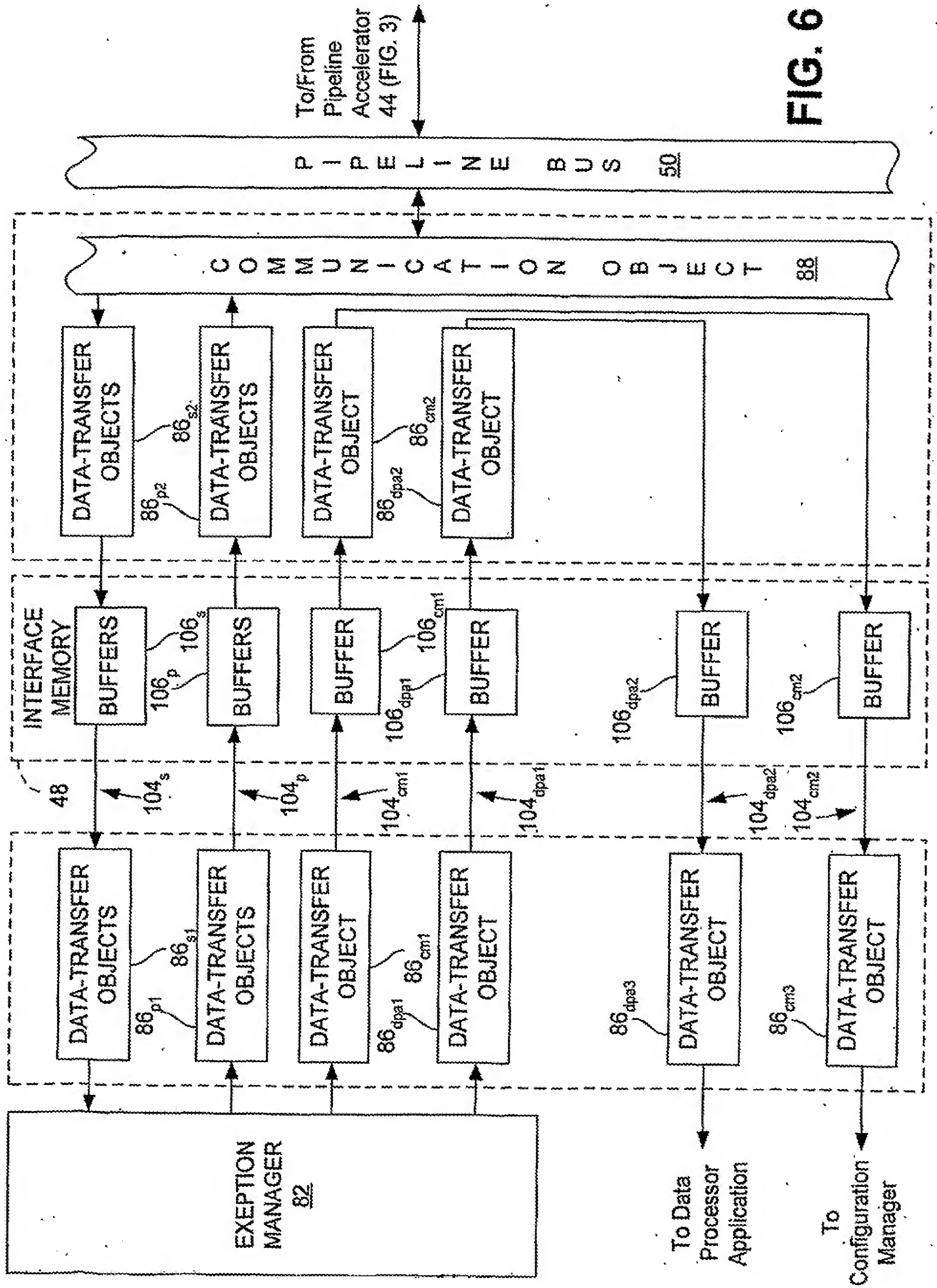


FIG. 5

SUBSTITUTE SHEET (RULE 26)



To/From Pipeline Accelerator 44 (FIG. 3)
PIPELINE BUS 50
COMMUNICATION OBJECT 88
DATA-TRANSFER OBJECTS
86_{s2}
86_{p2}
DATA-TRANSFER OBJECTS
DATA-TRANSFER OBJECT
86_{cm2}
86_{dpa2}
DATA-TRANSFER OBJECT
INTERFACE MEMORY
BUFFERS
106_s
106_p
BUFFERS
BUFFER
106_{cm1}
106_{dpa1}
BUFFER
106_{dpa2}
BUFFER
106_{cm2}
BUFFER
48
104_s
104_p
104_{cm1}
104_{dpa1}
104_{dpa2}
104_{cm2}
DATA-TRANSFER OBJECTS
86_{s1}
86_{p1}
DATA-TRANSFER OBJECTS
DATA-TRANSFER OBJECT
86_{cm1}
86_{dpa1}
DATA-TRANSFER OBJECT
86_{dps3}
DATA-TRANSFER OBJECT
86_{cm3}
DATA-TRANSFER OBJECT
EXEPTION MANAGER 82
To Data Processor Application
To Configuration Manager

FIG. 6

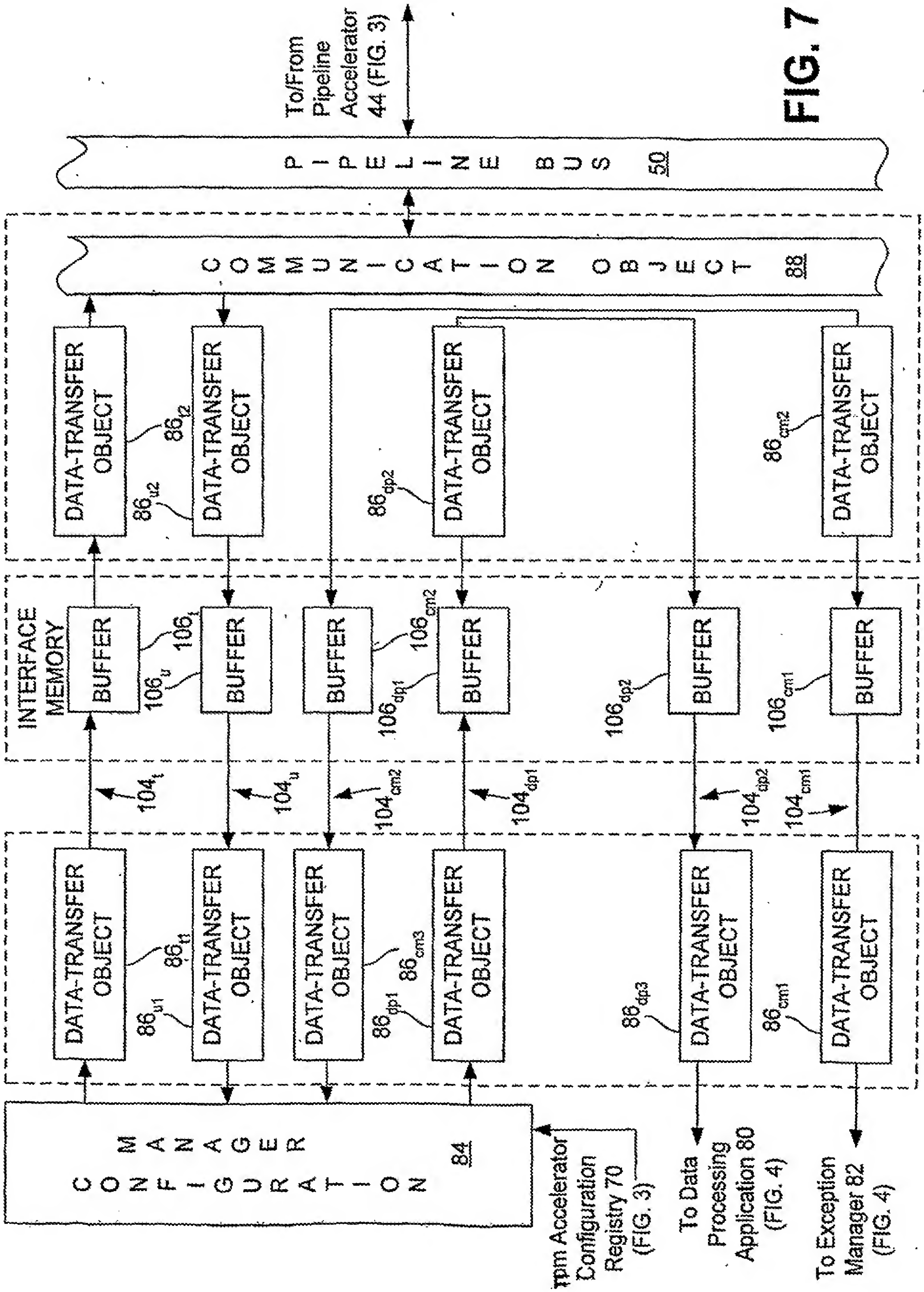

To/From Pipeline Accelerator 44 (FIG. 3)
PIPELINE BUS 50
COMMUNICATION OBJECT 88
DATA-TRANSFER OBJECT
86t2
86u2
86dp2
86cm2
INTERFACE MEMORY
BUFFER
106t
106u
106cm2
106dp1
106dp2
106cm1
104t
104u
104cm2
104dp1
104dp2
104cm1
86t1
86u1
86cm3
86dp1
86dp3
86cm1
CONFIGURATION MANAGER 84
From Accelerator Configuration Registry 70 (FIG. 3)
To Data Processing Application 80 (FIG. 4)
To Exception Manager 82 (FIG. 4)

FIG. 7